Abduction in Machine Learning

F. Bergadano[1], V. Cutello[1], and D. Gunetti[2]

[1]University of Catania, via A. Doria 6/A,
95100 Catania, Italy, bergadan@dipmat.unict.it

[2]University of Torino, corso Svizzera 185,
10149 Torino, Italy, gunetti@di.unito.it

Abstract

Abduction and induction are strictly related forms of defeasible reasoning. However, Machine Learning research is mainly focused on inductive techniques, leading from specific examples to general rules, with applications to classification, diagnosis and program synthesis.. Abduction has been used in Machine Learning, but its use was typically an aside technique, to be integrated or added on top of the basic inductive scheme. We discuss the general relation between abductive and inductive reasoning, showing that they solve different instantiations of the same problem. Then we analyze the specific ways abbdution has been used in Machine Learning. Two different cases are individuated: (1) abductive reasoning used in explanation-based learning systems as a heuristic to guide search in top-down specialization, and (2) abduction used for generating missing examples in relational learning. In both cases, the use of abduction is not general and adapted to a very tiny and specific problem. In this sense, the Machine Learning community has not used abduction as a synonym of induction, despite the high degree of similarity. However, both uses of abduction in learning have been proved to be effective for their intended purposes.

1 Introduction

Both inductive learning and abductive reasoning start from specific facts or observations and produce some explanation of these facts. Both may be described as forms of defeasible reasoning from effects to causes. There are some differences, but they are minor and due to different understandings of the notions of observation and explanation. In [1], such differences are analyzed in detailed, and some related discussion is also found in the next section or the present paper. To put it in one sentence, induction views examples as *instances* of a *concept* and explanations as general *concept descriptions*, while abduction views examples as specific observations and explanations as other specific facts that are true, and cause the observations to occur. As a typical case for induction, an example could be the description of a specific bird, and a concept description could be a rule such as

$$\text{bird(X)} \leftarrow \text{small(X)} \wedge \text{part\_of(X,Y)} \wedge \text{wing(Y)}$$

For abduction, an oservation could be the fact that some specific bird does not fly, and an explanation could be the fact that that specific bird has a broken wing. In both cases, the relevant logical relation is that the explanation, together with the domain knowledge, implies the observation.

The Machine Learning literature has largely ignored such similarities, or has produced studies that emphasize the differences that are present [8, 10]. On the other hand, abduction has been used as an effective technique *within* the underlying inductive framework of Machine Learning methods. We have identified two different approaches. First, abduction has been used to guide search in top-down relational learning. The idea has evolved from explanation-based learning methods, restricting the inductive hypotheses to be logical consequences of a given domain theory. This is then generalized by allowing inductive hypothesis to be obtained from the domain theory by either deductive or abductive reasoning. Second, abduction has been used to generated missing examples in relational learning. In fact, it is intrinsic to the nature of induction that the input data is incomplete: some examples are given, but not all. If some particular examples are missing, existing relational learning methods may however encounter serious problems, and abductive generation of these missing examples may be an effective solution. The following two paragraphs that conclude this introduction summarize these two ways of using abduction in learning. These two approaches are, we think, quite representative of the Machine Learning view of abductive reasoning.

Top-down relational learning algorithms suffer from the difficulty of serach a space of possible inductive hypotheses that is usually very large. In the first relational learning systems, and also in more recent approaches such as Foil [15], the problem was addressed with heuristics of a statistical nature. Top-down systems start from very general concept descriptions, and then try to obtain consistent rules with a number of specialization steps. Typically, heuristics would favor specializations that exclude negative examples while still covering a large number of positive examples. Such heuirstics may be misleading, and may also be insufficient for an adequate pruning of the hypothesis space. The Explanation-Based Learning (EBL) paradigm restricts the concept descriptions that may be possibly produced to the logic consequences of a so-called *domain theory*, that is given as an input to the learning system [11, 2, 14]. Abduction has been used in this framework as follows: given the domain theory, also the concept descriptions that may be obtained via abductive reasoning are considered as possible, and should be evaluated inductively on the basis of the available examples. Studies that follow this scheme may be found in [3, 6, 7].

A well known problem in Machine Learning is to provide a learning method with an "adequate" set of examples of the target concept. Here "adequate" informally means that the training set should contain all those examples required to successfully complete the learning task, and no more. Obviously, this is a very hard condition to achieve, since usually it is not possible to know in advance exactly which examples are (and which are not) significant for learning a concept. As a consequence, the learning task may turn out to be too slow (if too many examples are given) and/or fails (if the examples are not significant). This problem is particularly serious for an important class of learning methods: *Relational Learning Algorithms* based on an extensional interpretation of sub-predicates and recursion [15, 2, 14]. In these methods, the learning procedure can not only fail or be too slow, but also produce wrong results: the description of the target concept synthesized by the system may entail some of the negative examples given to the system. In this paper we show how abduction can be used to remedy the above problem. An abductive procedure is used to query the user for any example that may be missing, depending on the hypothesis space that has been defined and the given examples. A similar technique has been used before, for example in [16, 18], to query the user for missing values allowing a single example to be covered. The novelty of our approach is that abduction is systematically applied over the whole hypothesis space. As a result of this, the learning systems turns out to be correct and sufficient, in the sense that the learned description does not entail any of the negative examples and such a description can always be found if it exists.

2 Abduction and Induction

Abductive and Inductive Reasoning in Artificial Intelligence are considered to be distinct and have generated separate fields of study. After a simple analysis, one finds in effect distinct inference schemes. For induction:

$$\frac{\mathrm{P(a)}}{\forall x\ \mathrm{P(x)}}$$

For abduction:

$$\frac{\mathrm{P(x)} \rightarrow \mathrm{Q(x)},\ \mathrm{Q(a)}}{\mathrm{P(a)}}$$

Deeper analysis, however, suggests that the difference between the two schemes is not always easy to state. For instance, using the tautology $\Phi = (\forall x\ \mathrm{P(x)}) \rightarrow$

P(a) one gets:

$$\frac{\Phi,\ \mathrm{P(a)}}{\forall x\ \mathrm{P(x)}}$$

as an abductive inference step, but actually has the same premises and conlusions of the inductive inference scheme. It would then seem that the abductive scheme includes simple forms of inductive reasoning. In [1] we start from the above considerations and isolate some minor differences of abductive and inductive reasoning within the same franework, that is determined by the above inference rules. In the same paper, a formalization based on non-monotonic logic is developed. Here, we simply note that induction and abduction are indeed very similar inference schemes. However, the Machine Learning literature has not used them as synonyms. The main keyword in learning is *induction*, and abductive reasoning is rather used as an additional technique for solving particular problems. We discuss in the following sections two such uses of abduction in Machine Learning. In the next section we survey its use as a form of bias for guinding search in top-down specialization. In the rest of the paper, we show how abductive reasoning and queries can solve some relevant problems in relational top-down learning.

3 Guiding Search with Abduction

One use of the keyword *abduction* in the Machine Learning literature is related to the problem of guiding search in top-down specialization. As the number of concept descriptions that may possibly be generated is, in general, very large, and because even the number of descriptions that are consistent with the examples can be large, learning systems need extra-evidential criteria to prune the search space. Deductive inference with a domain theory has been used to this purpose in ML-SMART [2] and FOCL [14]. Similarly, a domain theory could be used abductively within the same framework as in [3] and in [6, 7].

Before we can suitably describe the details of abductive inference to prune the space of possible concept descriptions, we need to define a type of analytic learning called Explanation-based Learning (EBL), which has received much attention during the late eighties. A survey of EBL is found in [9].

EBL needs in input one or more positive examples, as well as a so-called *domain theory*, which includes relevant prior knowledge. A resolution proof of the positive examples is produced, and the leaves of the proof tree are generalized and taken as the antecedent of a new rule for the target concept. Suppose, for instance, that the domain theory is as follows

C(X) :- B(X), A(X,Y).
B(X) :- R(X,Y).
B(b).
A(c,a).
R(c,d).

and a positive example for C is C(c). This example can be obtained deductively from the domain theory, where the leaves in the proof tree are R(c,d) and A(c,a). This yields the learned description for C:

C(X) :- R(X,Y), A(X,Z)

The learned description is actually a logical consequence of the given domain theory, and could be obtained from the domain theory even without looking at the examples, by resolving the first two clauses in the domain theory. The role of the example is that of suggesting some consequences of the domain theory rather than others: the ones that are useful to deduce the positive example are chosen.

One natural question arises: what is the purpose of this form of learning, if its output is just a logical consequence of something that was already known?

The answer is that, although nothing really new is learned, the new form of knowledge may be more *operational*, i.e., easier to use, or leading to more efficient computations. In the above case, the positive example C(c) may be deduced more efficiently from the learned clause than by the original domain theory, as one resolution step is saved. EBL may then be seen as a form of pre-compilation, or partial evaluation. The effectiveness of this form of speed-up learning depends on whether the given positive examples are representative of future cases: if this is not true, the learned clause may turn out to be useless, as the original domain theory would be necessary every time. Learned clauses that are useless in this sense will only take away some memory space, without giving any computational advantage.

For this reason, systems soon started using EBL with many positive examples, so that only clauses that frequently proved to be useful woul be kept. Other systems (e.g., [2]) also introduced the possibility of using negative examples, in connection with the acceptance of a domain theory that might be partly incorrect. In this case, even clauses that follow deductively from the domain theory may be incorrect, and may cover negative examples. Clauses covering too many negative examples may then be discarded. When used in this way, the domain theory basically defines a hypothesis space. The concept descriptions that are logical consequences of the domain theory are descriptions that may potentially be learned. The description which is actually produced would also be required to perform well with respect to the data, i.e. cover many positive examples and few negative examples. EBL may then be considered as a way to guide search in top-down specialization: not all specializations are possible, but only the ones that produce deductive consequences of the domain theory. Legal specializations are obtained by resolving the clause to be specialized against some clause in the domain theory. Other specializations are not considered. In the previous example, the clause

C(X) :- R(X,Y), A(X,Z)

would be a legal specialization of the clause

C(X) :- B(X), A(X,Y).

while, e.g., the clause

C(X) :- B(X), A(X,Y), B(Y)

would not be considered. If the space of possible clauses is too large, a domain theory can be used effectively to prune this space and suggest prefered specialization steps. ML-SMART [2], and FOCL [14] adopt similar goals and techniques.

Abduction comes into place if the consequences of the domain theory are not just taken to be deductive consequences, but are also obtained by means of an abductive theorem prover. For instance, if the above domain theory would contain the clause

D(X) :- A(X,Y)

then, an abductive/deductive consequence of the theory would be the clause

C(X) :- R(X,Y), D(X)

which would also be a legal specialization of the clause

C(X) :- B(X), A(X,Y).

Such an abductive use of a domain theory is found in [3], with the purpose of restricting the hypothesis space without excluding concept descriptions that may be meaningful on the basis of abductive reasoning. Cohen introduces the notion of *abductive EBL* in a similar framework [6, 7].

4 Relational Learning Algorithms based on Extensionality

Relational Learning Algorithms learn recursive relational descriptions from positive and negative examples, given as ground literals. Usually, a subset of the first-order predicate calculus, Horn clauses, is used for the concept description (see, e.g. [5, 17, 6, 4, 12]). Learning definite clauses for multiple predicates is difficult, and systems tend to be slow. Many systems, such as Foil [15], Focl [14] and Golem [12] handle this problem evaluating clauses extensionally. In this way candidate clauses can be generated directly from the examples one at a time and independently of one another. The next subsection contains a simplified version of Foil, in order to illustrate the extensional approach and its drawbacks.

4.1 The Extensional Approach: Simplified Foil

Let P be the target concept and pos_examples(P) and neg_examples(P) the given positive and negative examples of P (in the following, α and γ represent generic conjunction of literals).

```
Simplified Foil:
while pos_examples(P) $\neq \emptyset$ do
    Generate one clause "P :- $\alpha$";
    pos_examples(P) $\leftarrow$ pos_examples(P) $-$ pos($\alpha$)
```

```
Generate one clause:
$\alpha \leftarrow$ true;
while pos($\alpha$) $\neq \emptyset$ do
    if neg($\alpha$) = $\emptyset$ then return(P :- $\alpha$)
      else choose a predicate Q and its arguments Args;
           $\alpha \leftarrow \alpha \wedge$ Q(Args)
```

where pos(α) and neg(α) are the sets of positive and negative examples of P covered by α. Every predicate Q can be defined by the user intensionally by means of logical rules or extensionally simply by giving some examples of its input-output behavior. In particular, clauses can be recursive and, in this case, Q = P, and its truth value can only be determined by the available examples.

Definition 1: We say that the clause P(X,Y) :- α(X,Y) *extensionally covers* an example P(a,b) iff:

- α = Q(X,Y). Then Q(X,Y) extensionally covers P(a,b) iff Q(a,b) is derivable from the definition of Q or is a given example of Q.
- α = γ(X,T), Q(T,Y) covers P(a,b) if there exists a value e such that the conjunction γ(a,e) is true and Q(e,b) is derivable from the definition of Q or is a given example of Q.

The choice of the literal Q(Args) to be added to the partial antecedent α of the clause being generated is guided by heuristic information. It might nevertheless be a wrong choice in some cases, in the sense that it causes the procedure "Generate one clause" to fail by exiting the while loop without returning any clause. This problem can be remedied by making the choice of Q(Args) a backtracking point.

Suppose, for instance, that Foil is given the following positive and negative tuples of the *ancestor* relation:

+ <r,g>,<b,g>,<b,d>,<d,g>,<b,r>,<r,d>,<r,p>;
− <d,d>,<g,p>,<d,p>.

where we also have an intensional definition for *parent*:

parent(X,Y) :- *mother*(X,Y)
parent(X,Y) :- *father*(X,Y)

where *mother* and *father* are (extensionally) defined by the following pairs of values:

mother			*father*	
X	Y		X	Y
d	g		f	g
r	d		s	d
r	p		s	p
i	r		b	r
t	s			
a	c		h	c

Finally, we know that the logic program for *ancestor* depends on *parent* and on itself (i.e. it may be recursive). As there are at most 3 variables to be used, these are the possible literals:

parent(X,Y), *parent*(Y,X), *parent*(X,W),
parent(W,X), *parent*(Y,W), *parent*(W,Y)
ancestor(X,W), *ancestor*(W,X), *ancestor*(Y,W), *ancestor*(W,Y) [1].

The learning algorithm starts to generate the first clause - the antecedent α is initially empty. We need to choose the first literal Q(Args) to be added to α. As we have left the heuristics unspecified, we will choose it so as to make the discussion short.

Let α=*parent*(W,Y); then all positive examples and the second and third negative examples are covered, so more literals need to be added.

Let α=*parent*(W,Y) $\wedge$ *parent*(W,X); in this case no positive examples are covered and the negative example <d,p> is covered. Clause generation fails and we backtrack to the last literal choice.

Let α=*parent*(W,Y) $\wedge$ *ancestor*(X,W); no negative example is covered, and the first 3 positive examples are extensionally covered. A clause is generated and the covered positive examples are deleted.

We proceed to the generation of another clause; α is empty again. If we choose the first literal as *parent*(X,Y), the remaining positive examples are covered and

[1] *ancestor*(X,Y) and *ancestor*(Y,X) are not listed because they may produce looping recursions

the final solution is obtained:

ancestor(X,Y) :- *parent*(W,Y), *ancestor*(X,W).
ancestor(X,Y) :- *parent*(X,Y).

4.2 Problems of extensionality

The independence of the clauses is made possible by the extensional interpretation of recursion and sub-predicates: when a predicate Q occurs in a clause antecedent α, it is evaluated as true when the arguments match one of the positive examples. For instance, the clause
ancestor(X,Y) :- *parent*(W,Y), *ancestor*(X,W).
was found to extensionally cover the example <b,g> of *ancestor* because *parent*(d,g) is true, and <b,d> is also a positive example of *ancestor*. The previously generated logical definitions of Q are not used. The method is (partially) justified by theorem 1 below.

Definition 2: A definition P is *complete* w.r.t. the examples E iff ($\forall$ Q(i,o) $\in$ E) P $\vdash$ Q(i,o). A definition P is *consistent* w.r.t. the examples E iff ($\nexists$ Q(i,o) $\in$ E) P $\vdash$ Q(i,o') and o$\neq$o'.

Theorem 1: [4] Suppose Foil successfully exits its main loop and outputs a logic program P, that always terminates for the given examples.
Let Q(X) :- α be a generated clause of P, then,
$\forall$q+$\in$Q α extensionally covers q+ $\rightarrow$ P $\vdash$ Q(q+).

However, extensionality forces us to include many examples, which would otherwise be unnecessary. In fact other desirable properties, similar to the one given by Theorem 1, are not true, and two fundamental problems arise:

Problem 1: For a logic program P, it may happen that P $\vdash$ Q(q+), but none of its clauses extensionally cover q+. As a consequence Foil would be unable to generate P. Consider this program P:

ancestor(X,Y) :- *parent*(W,Y), *ancestor*(X,W).
ancestor(X,Y) :- *parent*(X,Y).

Let <r,g> be the only positive example of *ancestor*. This example follows from P (P $\vdash$ *ancestor*(r,g)) but is not extensionally covered: the second clause does not cover it because *parent*(r,g) is false, and the first clause does not cover it extensionally because *parent*(d,g) is true, but <r,d> is not given as a positive tuple of *ancestor*.

Problem 2: Even if no clause of a definition P extensionally covers a negative example q- of Q, it may still happen that P $\vdash$ Q(q-). Therefore Foil might generate a definition that covers negative examples that were given initially (i.e. Foil is not consistent). Consider the following definition P:

ancestor(X,Y) :- *parent*(W,X), *ancestor*(W,Y)
ancestor(X,Y) :- *parent*(X,Y).

Then P $\vdash$ *ancestor*(g,p). Nevertheless, <g,p> is not extensionally covered by the first clause: *parent*(d,g) is true, but <d,p> is not a positive tuple of *ancestor*. Since d is not an ancestor of p, it could not possibly be added as a positive tuple, and <g,p> would not be extensionally covered even if all positive tuples were given. The solution differs from the one of problem 1: in this case P will be ruled out by adding a *negative* example, namely <d,p>.

5 Problems of Intensional Methods

Giving up the extensional interpretation of predicates while keeping the basic computational structure of Foil produces insuperable problems. In fact, the truth value of the missing examples could be obtained by means of the partial program generated at a given moment. But if the inductive predicates occurring in a clause being generated are evaluated by means of the clauses that were learned previously, then the order we learn these clauses with becomes a major issue.

Suppose, for instance, that we are given the following family tree:
parent(i,r), *parent*(r,d), *parent*(d,g), *parent*(f,g)
and the following positive and negative examples of *ancestor*:
+: <i,d>, <i,r>, <i,g>; −: <i,f>.
Suppose also that the following clauses are generated, with the given order:

(1) *ancestor*(X,Y) :- *parent*(X,W), *parent*(W,Y).
(2) *ancestor*(X,Y) :- *parent*(Y,W), *ancestor*(X,W).
(3) *ancestor*(X,Y) :- *parent*(X,W), *ancestor*(W,Y).

Clause (1) does not use any inductive predicate and immediately covers the positive tuple <i,d>. Clause (2) contains *ancestor*(X,W), and this literal is evaluated with the clauses available at this stage, i.e. (1) and (2). With this kind of intensional interpretation, the only tuple covered is <i,r>, a positive example. When clause (3) is generated, it will cover the positive tuple <i,g>. But clause (2) will now cover the negative tuple <i,f>, since *parent*(f,g) is true, and *ancestor*(i,g) may now be deduced by using the third clause.

Rules that seem consistent and useful at some stage may later be found to cover negative examples. We must then backtrack to the generation of the clause causing the inconsistency, e.g. clause (3), and to the generation of the clause that was later found to be inconsistent, e.g. clause (2). In general, we must abandon our former assumption that clauses may be learned one at a time and independently. Alternatively, we may number the predicates occurring in the learned clauses every time an inconsistency is detected. For instance, the former program would be rewritten as

*ancestor*1(X,Y) :- *parent*(X,W), *parent*(W,Y).
*ancestor*2(X,Y) :- *parent*(Y,W), *ancestor*2(X,W).
*ancestor*3(X,Y) :- *parent*(X,W), *ancestor*3(W,Y).
ancestor(X,Y) :- *ancestor*2(X,Y).   *ancestor*2(X,Y) :- *ancestor*1(X,Y).
ancestor(X,Y) :- *ancestor*3(X,Y).   *ancestor*3(X,Y) :- *ancestor*1(X,Y).

Nevertheless, this technique does not totally avoid the need of backtracking, and explodes the number of possible clauses by multiplying the number of inductive predicates by the number of generated indexes. A solution based on abduction is presented in the next subsection, so that the advantages of extensionality are preserved, i.e. so that previously generated clauses do not need to be reconsidered.

6 Completing examples via abduction before learning

There is no reason why particular positive (problem 1) or negative (problem 2) examples should have to be present. After all, the whole motivation of induction is that some information is missing. The important points are that (1) if a definition P consistent with the given examples exists, then it must be found and (2) the induced definition P must not cover any negative examples. The extensional approach guarantees neither, unless some specially determined positive and negative examples are given.

Here we show how abduction can be used to query the user for the missing examples., in order to preserve the computational advantages of extensional approaches, while guaranteeing that a correct solution be found. Suppose that a learning system has to cover the positive example P(a,b), and that the following candidate clause has been generated:

P(X,Y) :- α(X,Y), Q(Y,Z).

Moreover, we know that α(a,b) is true (for example because every literal in α is extensionally evaluated to true). Then, that clause will extensionally cover P(a,b) only if there exists some value c such that Q(b,c) is a positive example (known to the system) of Q. Suppose that all such examples are missing. From the classical abductive inference rule:

$$\frac{\alpha \leftarrow \beta \qquad \alpha}{\beta}$$

we get:

$$\frac{\text{P(a,b) :- }\alpha\text{(a,b), Q(b,Z).} \qquad \text{P(a,b)}}{\text{Q(b,Z)}}$$

and then the user can be queried for a value of Z such that Q(b,Z) is true. the new example of Q is added to the set of positive examples and P(a,b) can now be covered. This is a *controlled* form of abduction, in the sense that the result of abductive inference is not asserted as true, but only proposed as a possible truth, which is then queried to the user.

On the ground of this example, we have devised the following strategy. Every legal clause (= permitted by the constraints) of the type
P(X,Y) :- A(X,W), Q(Y,W,Z), α.
where Q is processed with the following abductive completion procedure:

```
for every example <a,b> of P do
    evaluate A(a,W), obtaining a sequence $W_1, ..., W_n$
        of partially instantiated answers.
    for every $W_i$ do
        ask the user for all the positive and
        negative examples of Q that match Q(b,$W_i$,Z)[2];
        add these examples to the positive examples of Q.
```

Adding one example may cause the request of others. Suppose, for instance, that a tuple <a,c> is added to the positive examples for A. Then, the above procedure

[2]Therefore, the procedure will succeed only if Q(b,W_i,Z) has a finite number of positive and negative instantiations

might add more positive or negative tuples of Q, e.g. the ones matching Q(b,c,Z). As a consequence, the procedure must be repeated for every clause, again and again, until no more examples are added.

Consider, for instance, the example about the *ancestor* relation given above, and the following two clauses:
relative(X,Y) :- *ancestor*(X,Y).
ancestor(X,Y) :- *parent*(W,Y), *ancestor*(X,W).
where <b,g> is the only positive example of *relative*, <d,d> is the only positive example of *relative*, and <f,f> is the only negative example of *ancestor*; by using the first clause, the user is queried for *ancestor*(b,g), and this is added to the positive examples of *ancestor*. Using the second clause, *parent*(W,g) answers W_1=f and W_2=d; the user is then queried for the truth value of *ancestor*(b,d) and *ancestor*(b,f), that get added to the positive and negative examples, respectively. Since f has no parents, <b,f> does not cause the addition of more examples. By contrast, the second clause can be used again with X=b and Y=d, and, after some repetitions of this procedure, we obtain the completed set of examples for *ancestor*:

+ <b,g>,<b,d>,<b,r>

− <f,f>,<b,f>,<b,i>,<b,s>,<b,t>.

Not all possible examples have been added, only the ones that were useful for those two clauses, given the initial examples. If this is done for all the clauses that are possible a priori, i.e. that satisfy the given constraints, then problems 1 and 2 do not longer hold:

Theorem 2: Suppose the examples given to an extensional learning system are completed with the above given abductive procedure. In this case, if the learning system successfully exits its main loop and outputs a definition P for a concept Q, then

P $\vdash$ Q(q) $\rightarrow$ q is extensionally covered.

Proof (by contradiction)

Suppose that (1) P $\vdash$ Q(q) but (2) q is not extensionally covered.

Let Q(X) :- β(X,Y) $\wedge$ R(Y) $\wedge$ γ be the clause resolved against Q(q). Suppose that P $\vdash$ β(q,r) $\wedge$ γ and P $\vdash$ R(r), but no such r is a positive example of R. There must be one literal R(Y) having this property, because of assumptions (1) and (2). Since the tuples of R must have been completed with the given procedure, the user has been queried for R(r), and this must have been inserted as a negative example. Therefore, it cannot be extensionally covered. We could now repeat the same argument for R. This would produce a non-terminating chain of resolution steps, and a finished proof of Q(q) would never be obtained, contradicting the hypothesis that P $\vdash$ Q(q).

Corollary 3: If the examples given to an extensional learning system are completed with the above given abductive procedure, and the learning system successfully exits its main loop and outputs a definition P, then P is consistent.

It should be noted that this abductive completion is done as a preprocessing step. Then, it guarantees that a solution consistent with the examples is found if it exists. Moreover, it does not require reconsideration of previously generated clauses, as do systems (e.g. MIS [18]) that ask for new examples only when they are needed and during the learning process.

7 Conclusion

The relation between induction and abduction is briefly discussed in this paper. Our main goal was then to show the specific uses of abductive reasoning in Machine Learning. On the one hand, it has been used to guide search in a top-down specialization framework related to Explanation-based Learning. On the other hand, we have also shown how abduction can be used to query the user for examples that may be missing. This means that the user must not provide all the needed examples to learn one definition at a time. He or she can forget some examples, and the abductive procedure will ask for them. Observe that only the examples really needed are queried, so it will not waste time trying to cover useless examples. In many extensional systems [12, 15] the user must provide all the examples at one time, and usually a superset of the examples needed is given, resulting in a lot of time wasted in covering useless examples.

Acknowledgement: The authors thank esprit project DRUMS II which is currently supporting part of this research.

References

[1] F. Bergadano and P. Besnard. Abduction and Induction by Non-Monotonic Logics, *Workshop on Mathematical and Statistical Methods in Artificial Intelligence*, Udine, Italy, September 1994.

[2] F. Bergadano and A. Giordana. A Knowledge Intensive Approach to Concept Induction. *Proceedings of the Fifth International Conference on Machine Learning.* pages 305–317, Ann Arbor, MI, 1988.

[3] F. Bergadano, A. Giordana, and S. Ponsero. Deduction in Top-down Inductive Learning, *Proc. of the sixth Int. Conf. on Machine Learning.* pages 23–25, Ithaca, N.Y., 1989.

[4] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Proc. Int. Joint. Conf. on Artificial Intelligence*, Chambery, France, 1993. Morgan Kaufmann.

[5] L. Birnbaum and G. Collins (Eds.). *Proc. Int. ML conference, part VI: Learning Relations.* Morgan Kaufmann, 1991.

[6] W. Cohen. Abductive Explanation-Based Learning: a Solution to the Multiple Inconsistent Explanation Problem. *Machine Learning*, 8:167–219, 1992.

[7] W. Cohen. Incremental Abductive Explanation-Based Learning. *Machine Learning*, 15:5–24, 1994.

[8] L. Console and L. Saitta. Generalization in Learning and Abduction. technical report, University of Torino, 1994.

[9] T. Ellman. Explanation-Based Learning: a Survey of Programs and Perspectives. *ACM Computing Surveys*, 21:2, pages 163-222, 1989.

[10] P. Flach. Simply Logical, John Wiley and sons, 1992.

[11] T. Mitchell and R. M. Keller and S. Kedar-Cabelli. Explanation-Based Generalization: a Unifying View, *Machine Learning*, 47-80, 1986.

[12] S. Muggleton and C. Feng. Efficient Induction of Logic Programs. In *Proc. of the first conf. on Algorithmic Learning Theory*, Tokyo, Japan, 1990.

[13] M. Pazzani and C. A. Brunk and G. Silverstein. A Knowledge-intensive Approach to Learning Relational Concepts. *Proc. of the 8th Int. Conf. on Machine Learning*, 1991.

[14] M. Pazzani and D. Kibler. The Utility of Knowledge in Inductive Learning. *Machine Learning*, 9:57–94, 1992.

[15] R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5:239–266, 1990.

[16] L. De Raedt and M. Bruynooghe. CLINT: a Multistrategy Interactive Concept-Learner and Theory Revision System. In *Proc. Workshop on Multistrategy Learning*, pages 175–190, 1991.

[17] C. Rouveirol, editor. *Proc. of the ECAI Workshop on Logical Approaches to Learning*. ECCAI, Vienna, Austria, 1992.

[18] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.